Vrije Universiteit Amsterdam

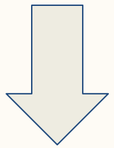
Testing in Kubernetes: A use case study of JetBrains CodeCanvas

He (Paige) Wen
Master Computer Science
VU Amsterdam

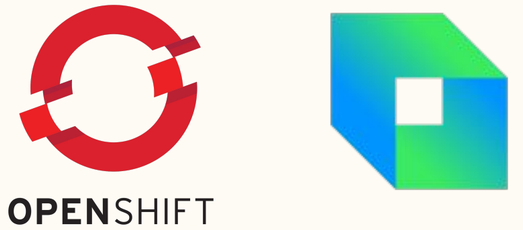
Supervisor
Daniele Bonetta (@Large Research)
Matthijs Jansen (@Large Research)
Alexander Chumakin (@JetBrains)

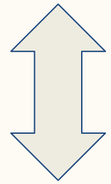
Introduction

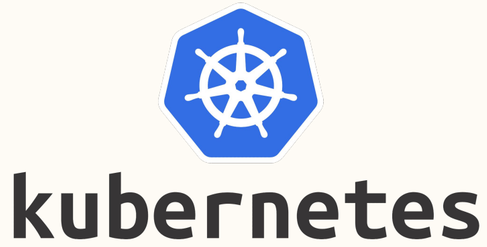
- Digital services ➔ distributed ➔ containerization
- **Kubernetes** popularity (80% production usage [1])
- Large systems that **integrate** with Kubernetes
  - deployed in Kubernetes
  - tight infrastructure integration (custom logic)
  - under rapid continuous development cycle



How to test these systems?







Problem Statement

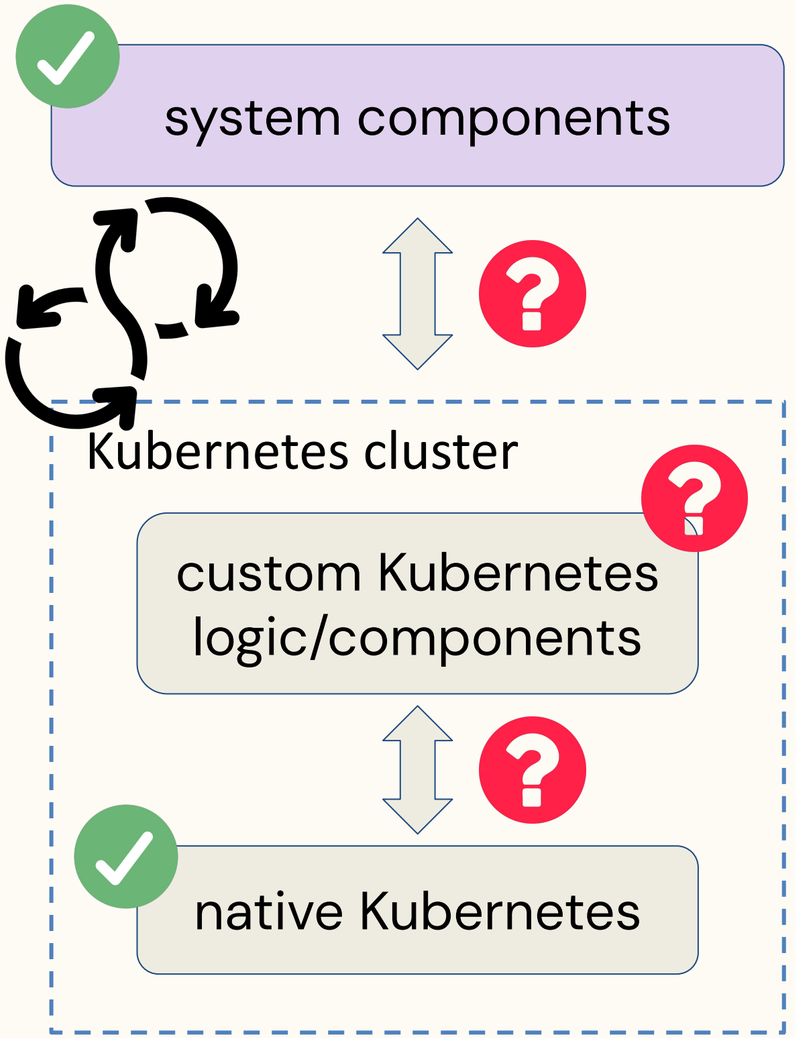
Research on **testing in Kubernetes:**

- testing application-level behavior [2,3]
- testing one single component inside Kubernetes [3,4]

narrowly-focused → data loss / incompatibility

There currently exists no **systematic methodology** or **automated framework** for **end-to-end** testing of Kubernetes-integrated systems that are both:

- applications built on Kubernetes,
- have **orchestration** logic integrated with Kubernetes control plane.



Research Questions

RQ1: How can we design a continuous orchestration-aware **testing framework** for Kubernetes-integrated systems?

RQ2: How can a comprehensive **test suite** be constructed for Kubernetes-integrated systems?

- **RQ2.1:** Which **components** of a Kubernetes-integrated system must be tested?
- **RQ2.2:** What are the **expected behaviors and outcomes** for different Kubernetes-based system components under various conditions?

RQ3: How can the **effectiveness** of the designed testing framework and test suite be evaluated?

Contributions

RQ1: Design a **testing framework**

RQ2: Design a **test suite**

C1: A **methodology** in end-to-end testing framework and suite design for Kubernetes-integrated systems.

C2: **Design & implementation** of a continuous, orchestration-aware **testing framework** for Kubernetes-integrated systems.

C3: **Design & implementation** of a end-to-end **test suite** for Kubernetes-integrated systems.

RQ3: Evaluate **effectiveness**

C4: Quantitative **evaluation** of the testing framework and test suite.

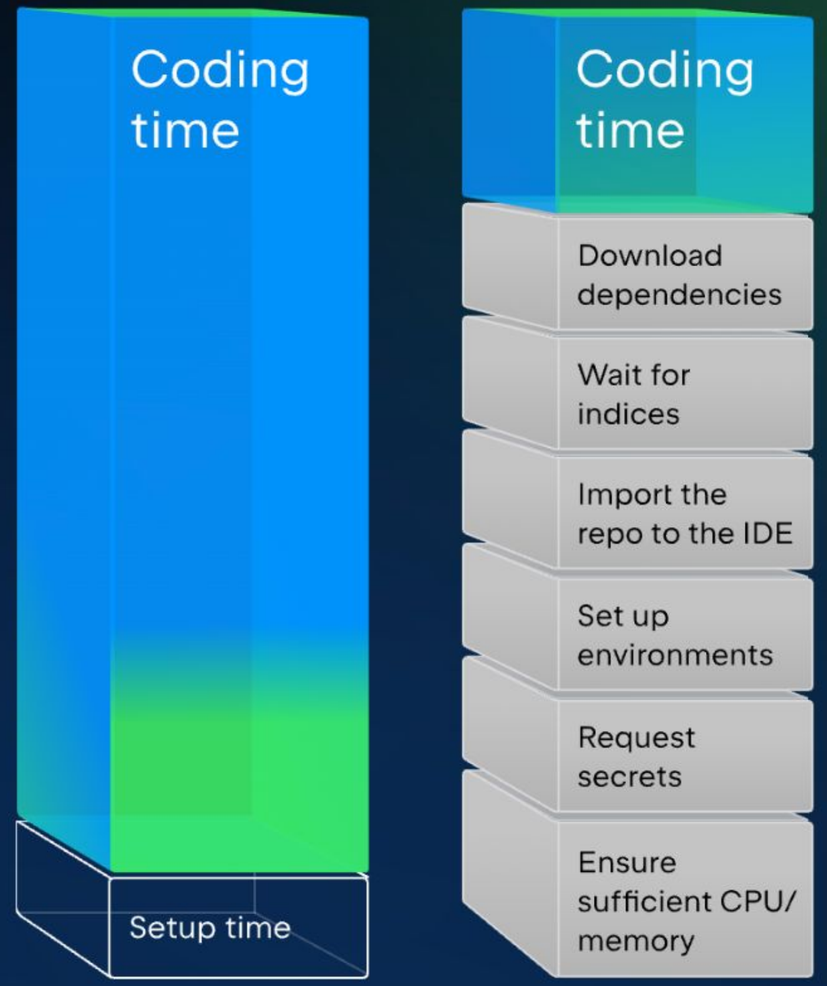
Use Case: JetBrains CodeCanvas

CodeCanvas

Less setup. More coding.

Cloud Development Environment Solution

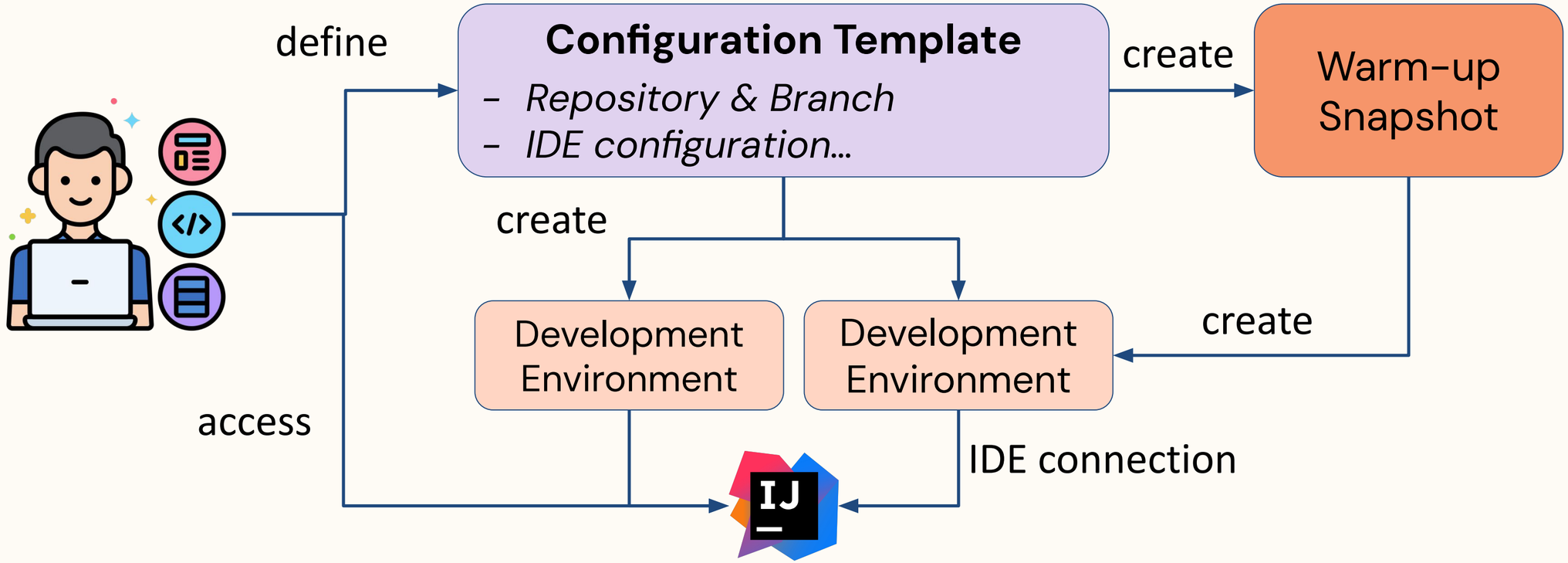
CodeCanvas is a cloud development environment (CDE) solution that speeds up your development, brought to you by the IDE professionals at JetBrains. Enjoy ready-to-use and cost-efficient development environments with your favorite IDEs within your own infrastructure.

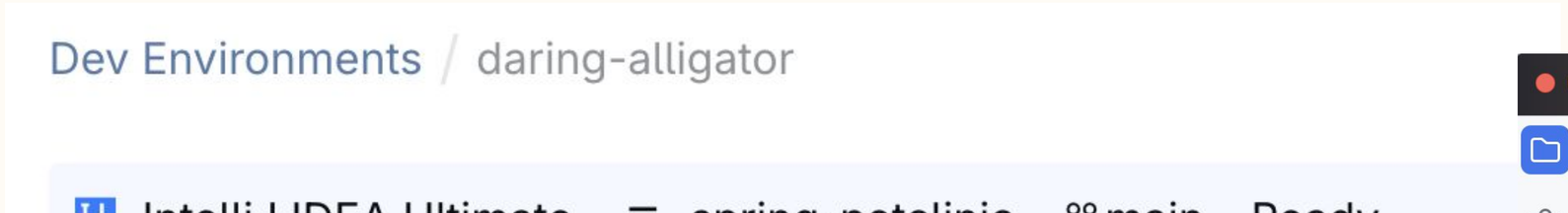


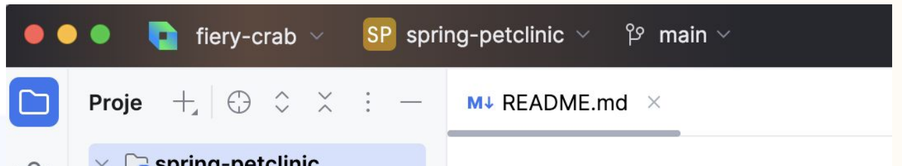
With CodeCanvas

Without

Background: **CodeCanvas**

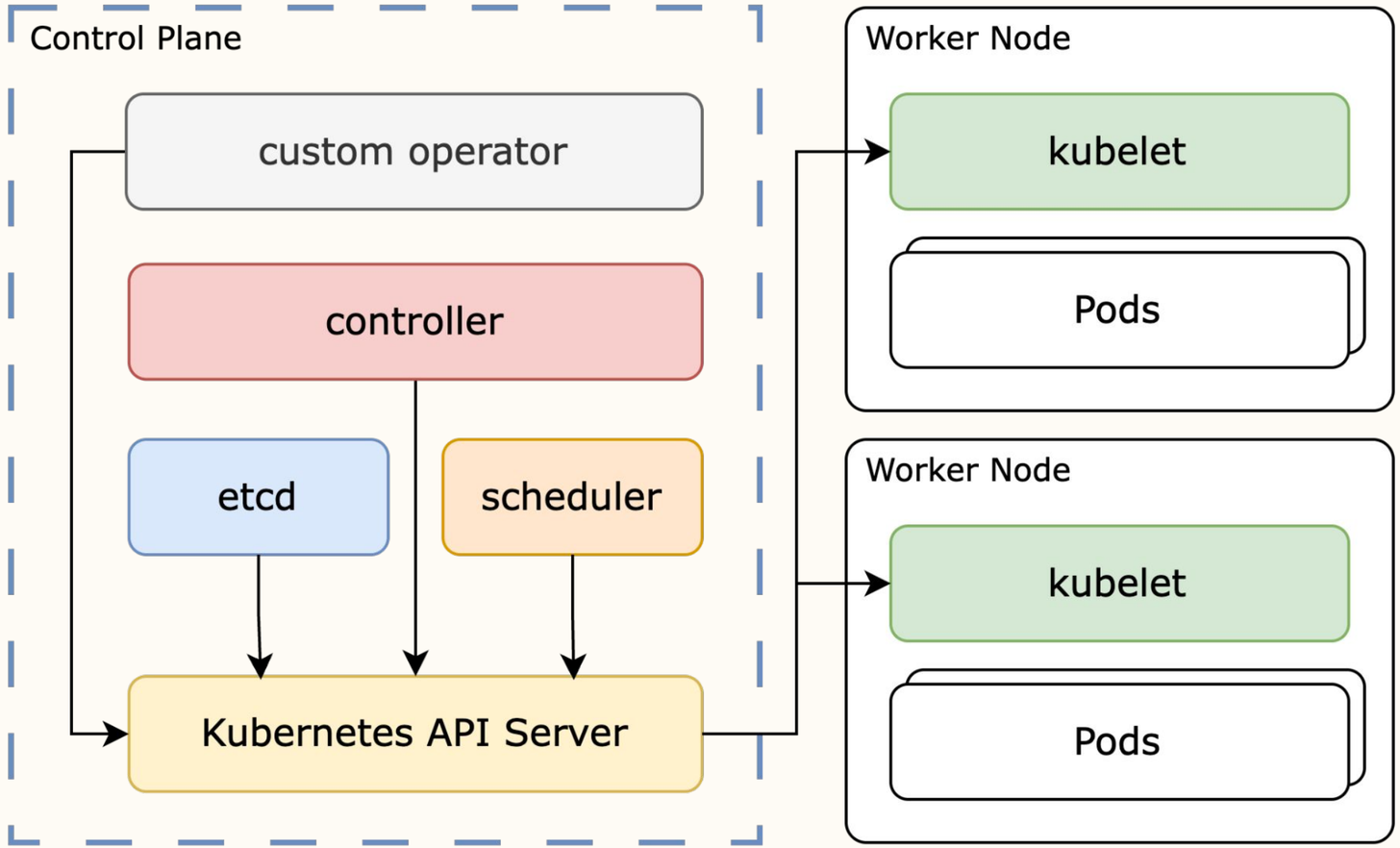






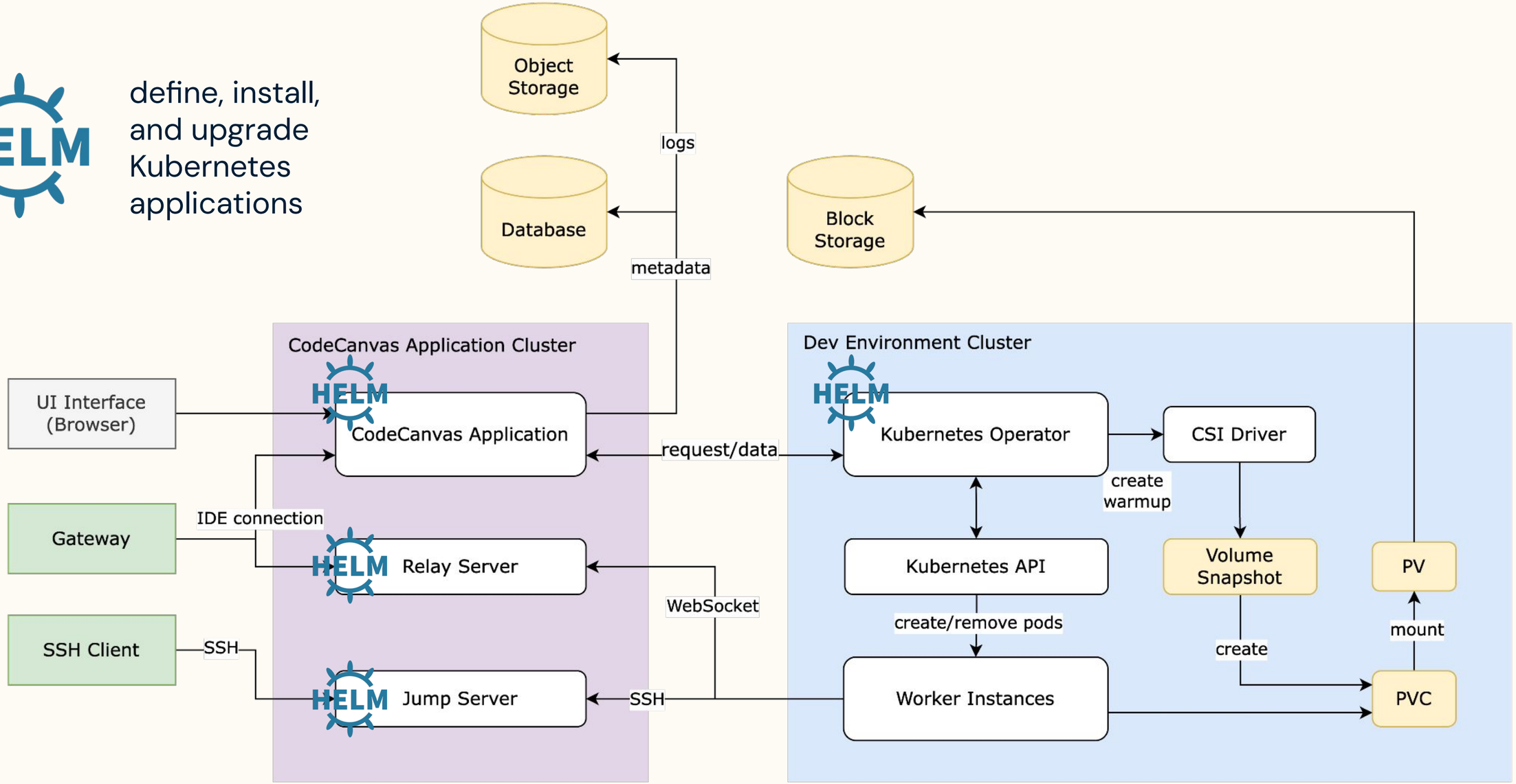
Background: **Kubernetes Architecture**

- **Controller:** monitor & control states (simple logic)
- **Operator:** use custom resources to manage applications (complex behavior)
- **etcd database:** stores cluster data
- **kubelet:** worker node agent



Background: **CodeCanvas Architecture**

define, install, and upgrade Kubernetes applications



Framework Design: What & How

System Workflow Analysis:

➔ Dev Environment Lifecycle

➔ Warm-up Snapshots

Involved Components
+
Orchestration Logic

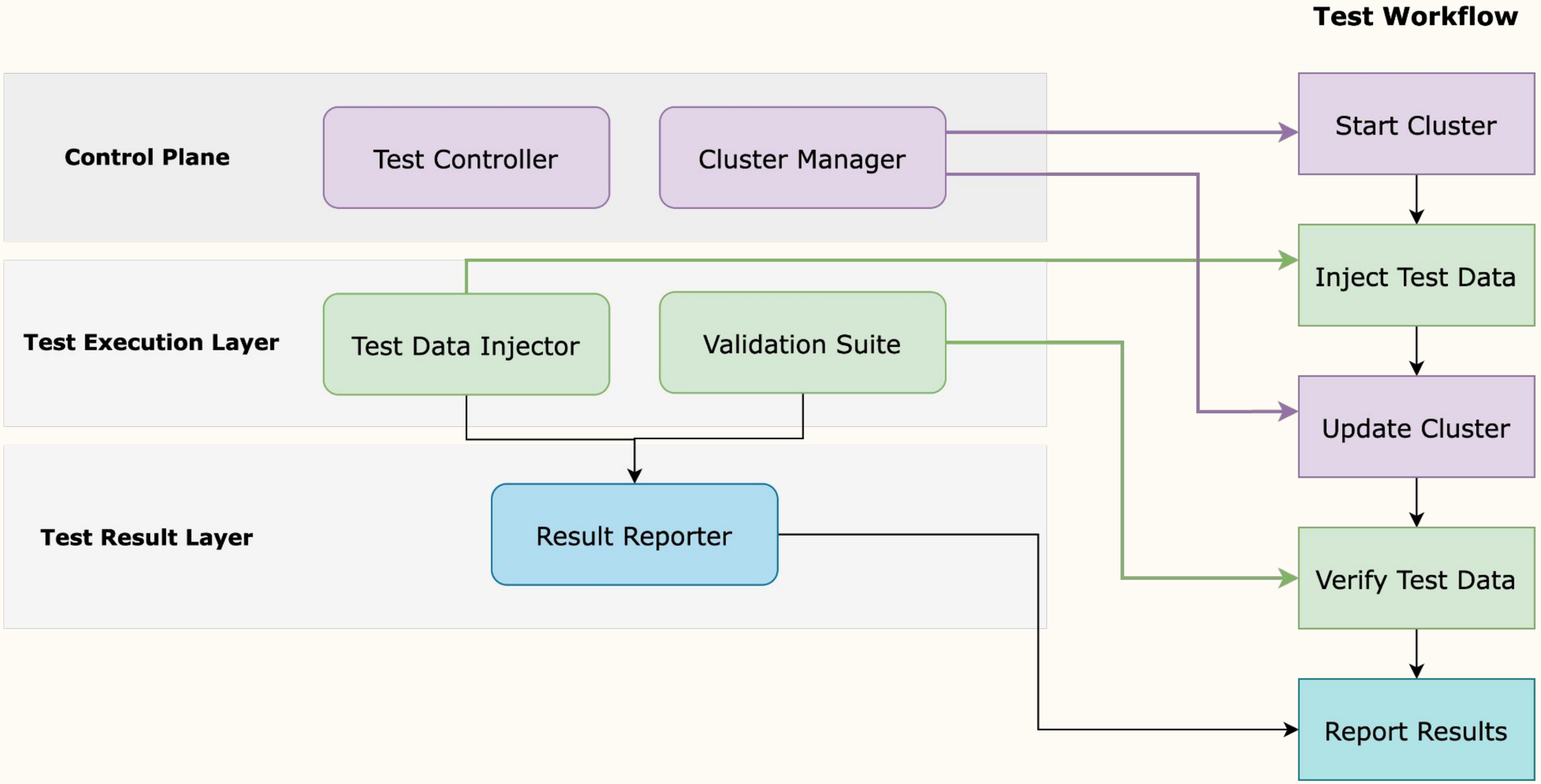
- Multi-layer interaction coverage
- Simulate product update
- Cluster lifecycle management
- Data Injection

Requirements Analysis:

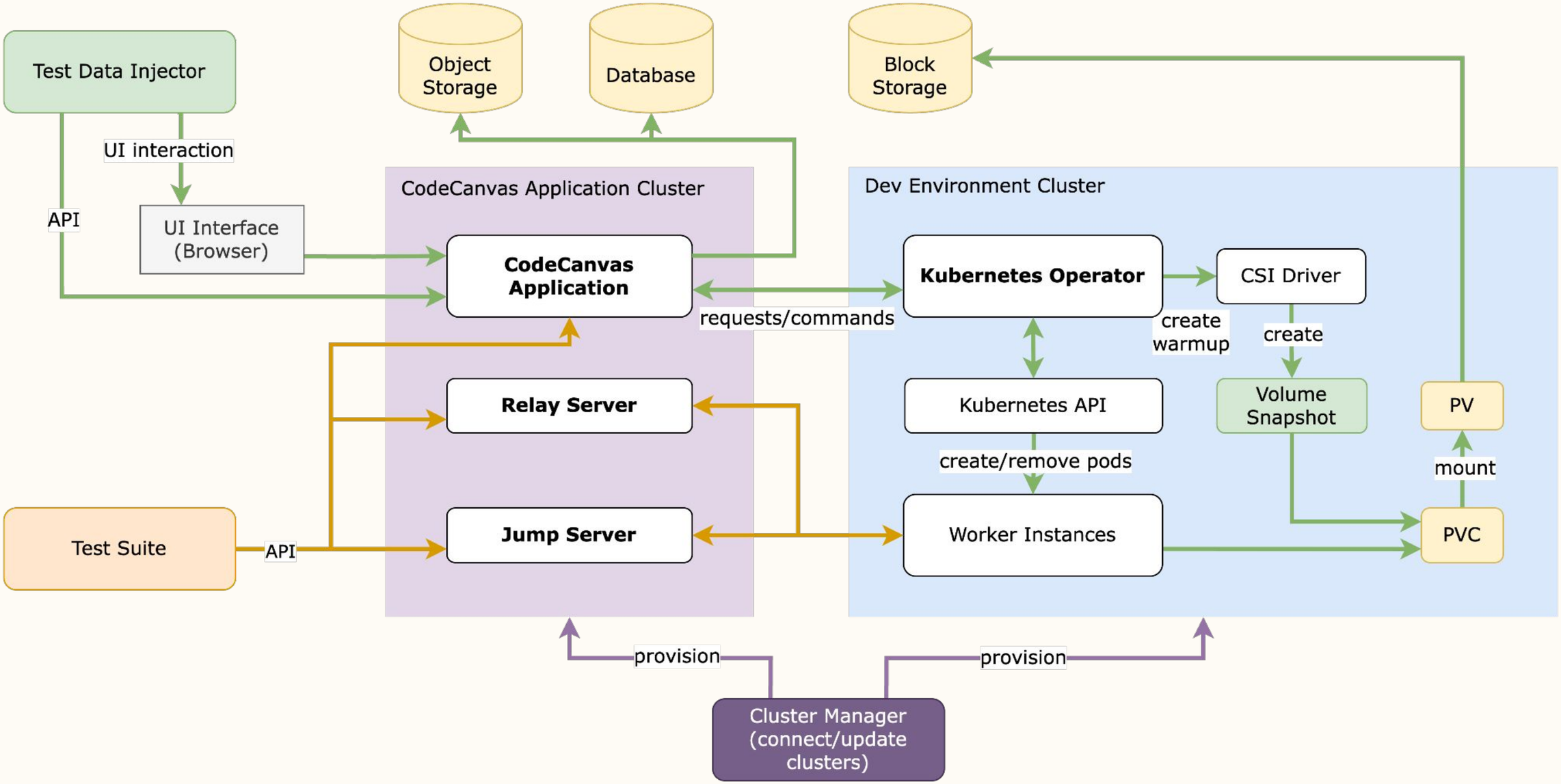
➔ Functional Requirements

➔ Non-Functional Requirements

Framework Design: Result



Orchestration-Aware Mechanism



Test Suite Design:
3-Step Methodology

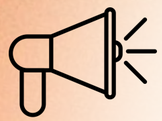
What does the system do?

A **component analysis** of the system under test.

What needs to be tested?

Identification of **Validation Goals**.

Test Case Scenarios

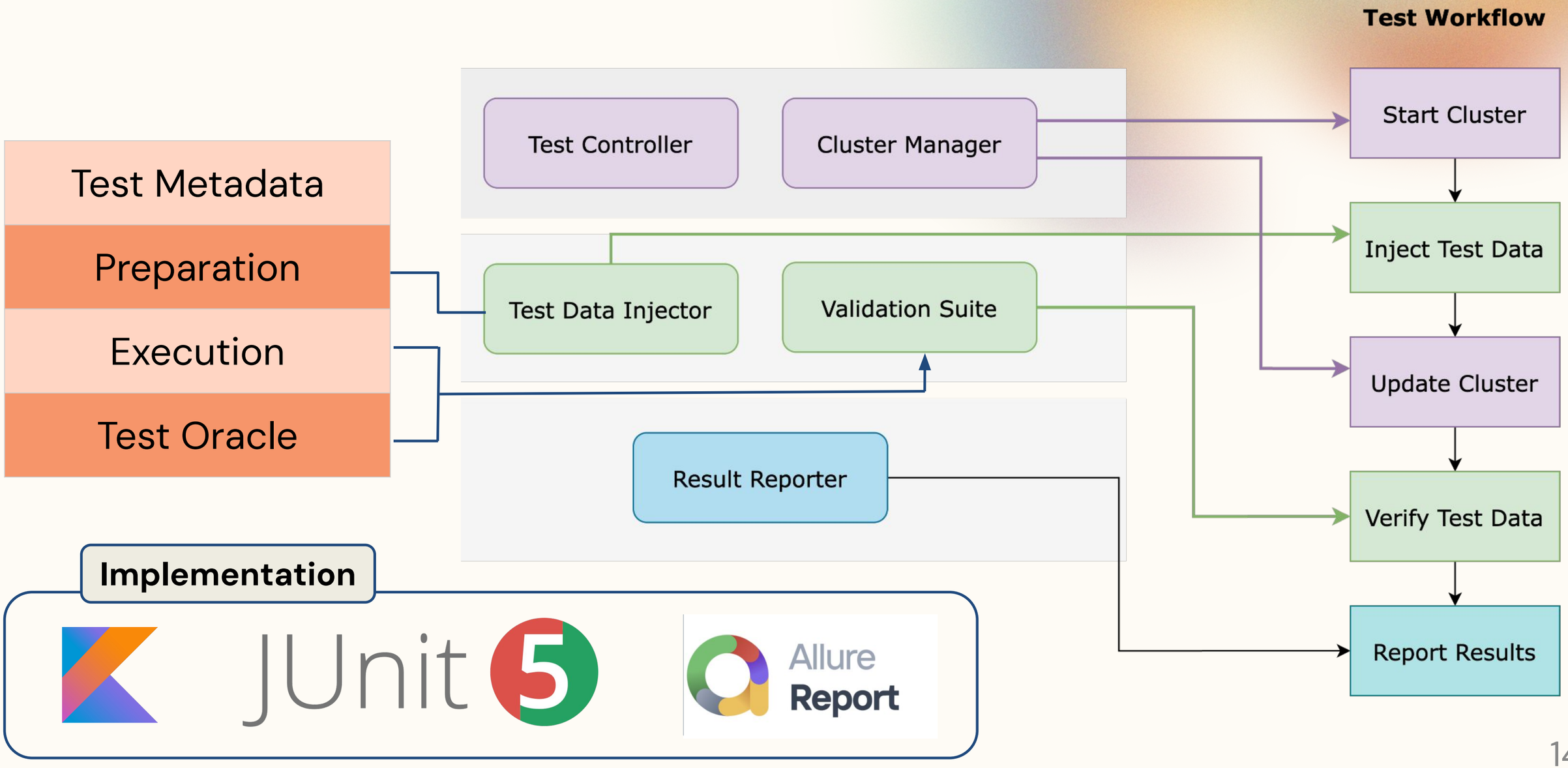


How to do the testing?

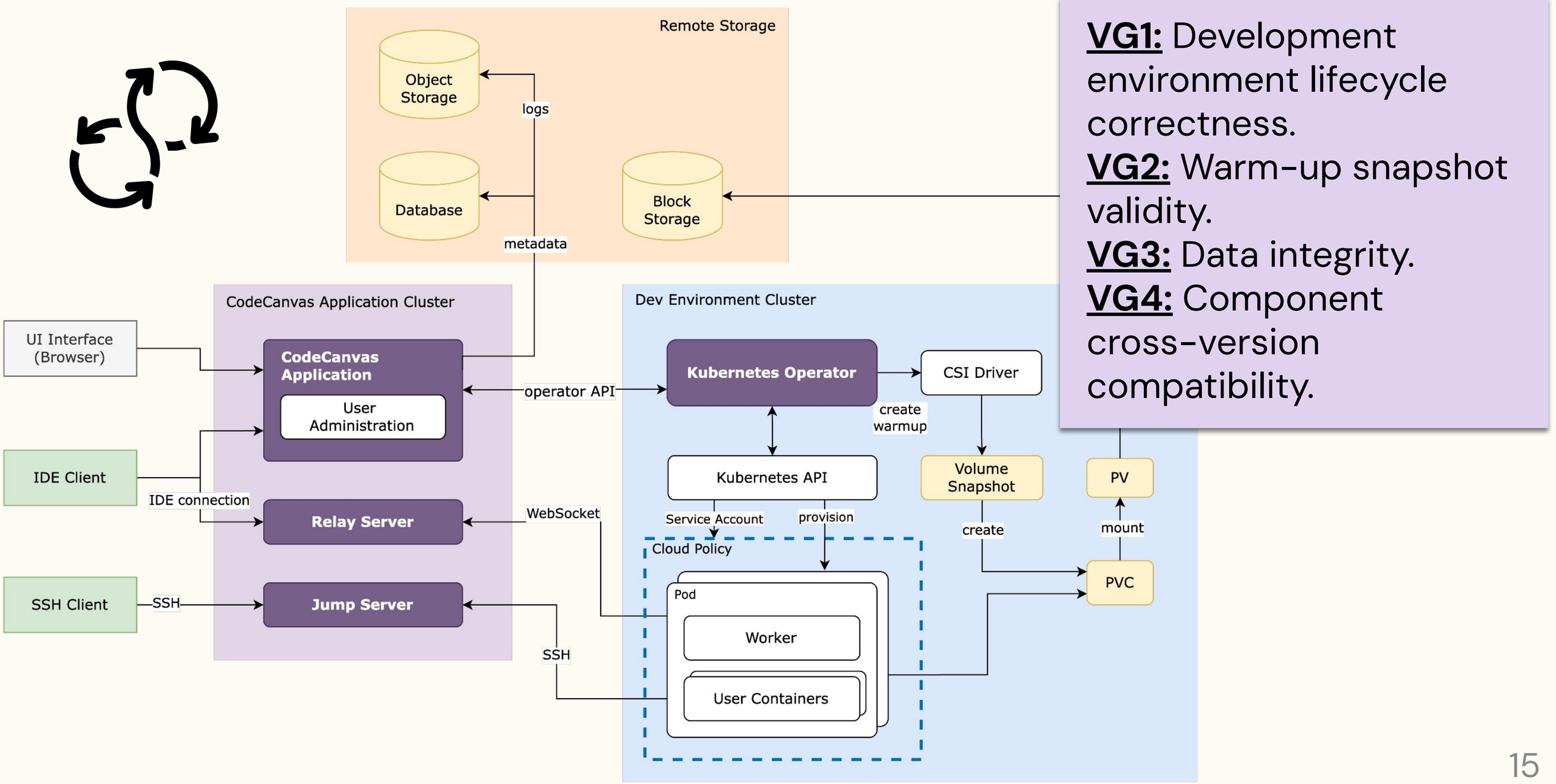
Test suite **Design Principles**.

Test Suite Structure

Test Suite Structure



Component Analysis & Validation Goals



VG1: Development environment lifecycle correctness.
VG2: Warm-up snapshot validity.
VG3: Data integrity.
VG4: Component cross-version compatibility.

VGs ➔ Use Cases?

VG1: Development environment lifecycle correctness.

- Environment provisioning, initialization, runtime behavior, and teardown. (core components)

VG2: Warm-up snapshot validity.

- Generated correctly, capture correct system state, can be restored. (CSI driver)

VG3: Data Integrity.

- Data in remote storage remain consistent and accessible after cluster update. (PV-PVC binding, database access)

VG4: Component cross-version compatibility.

- Newly deployed components compatible with old version components.

Use Case 1: Component Functionality Correctness

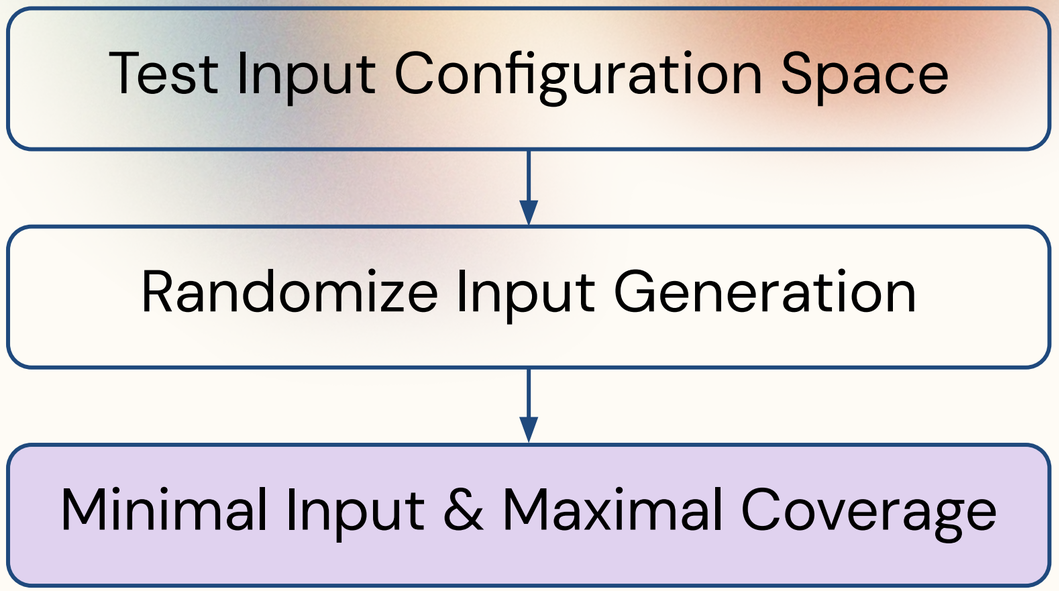
Use Case 2: Data Integrity

Use Case 3: Cross-Version Component Compatibility

Use Case ➔ Test Cases?

Test Input: Dual-Coverage Strategy

- Structural coverage
  - Components & workflow
  - Final State Machine (FSM)
- Operator command coverage
  - Command x Resource matrix

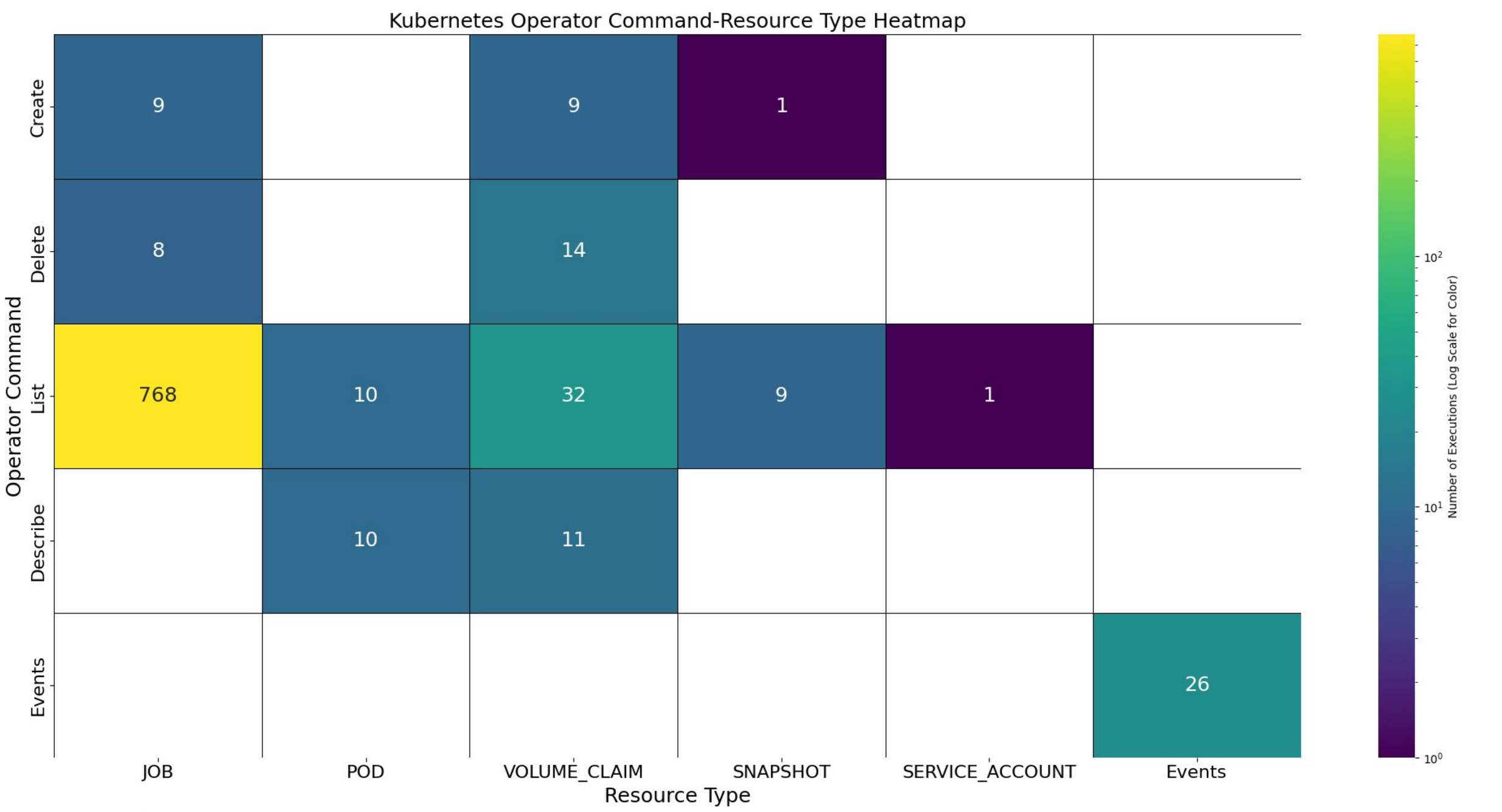


Test Oracle

- FSM transitions
- End-to-end workflows
- Kubernetes behaviors

Table 4.3: Test cases of component functionality co

ID	Description	Preparation	Execution
TC-01	New warm-up and dev. env. creation.	Create a template configuration and	Recreate warr template and


Kubernetes Operator Command-Resource Type Heatmap
Operator Command
Create
Delete
List
Describe
Events
9
9
1
8
14
768
10
32
9
1
10
11
26
JOB
POD
VOLUME_CLAIM
SNAPSHOT
SERVICE_ACCOUNT
Events
Resource Type
Number of Executions (Log Scale for Color)
10^2
10^1
10^0

Evaluation

UC1: **3** test cases

UC2: **8** test cases

UC3: **5** test cases

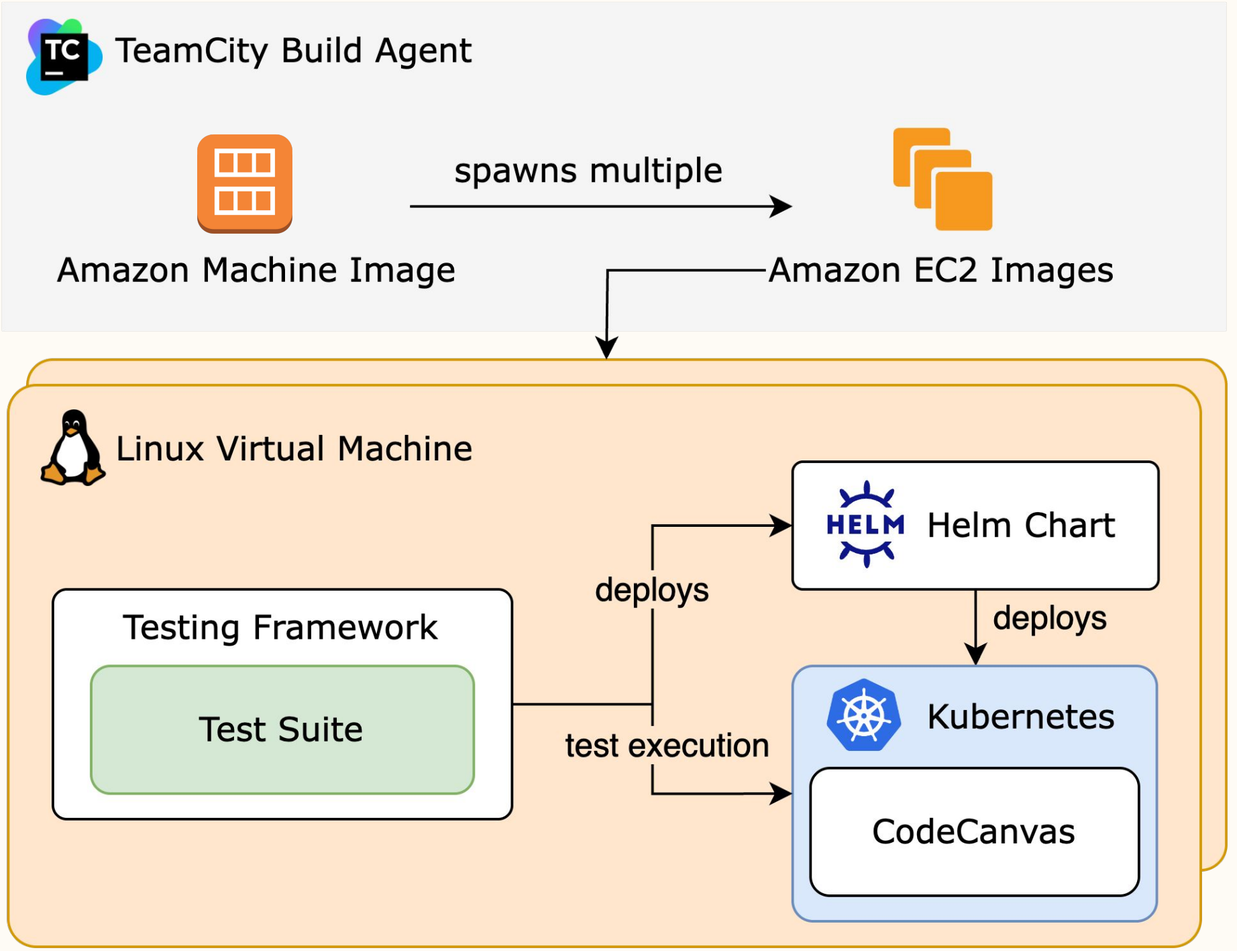
---

5 runs for each test case (rule out randomness / flakiness)

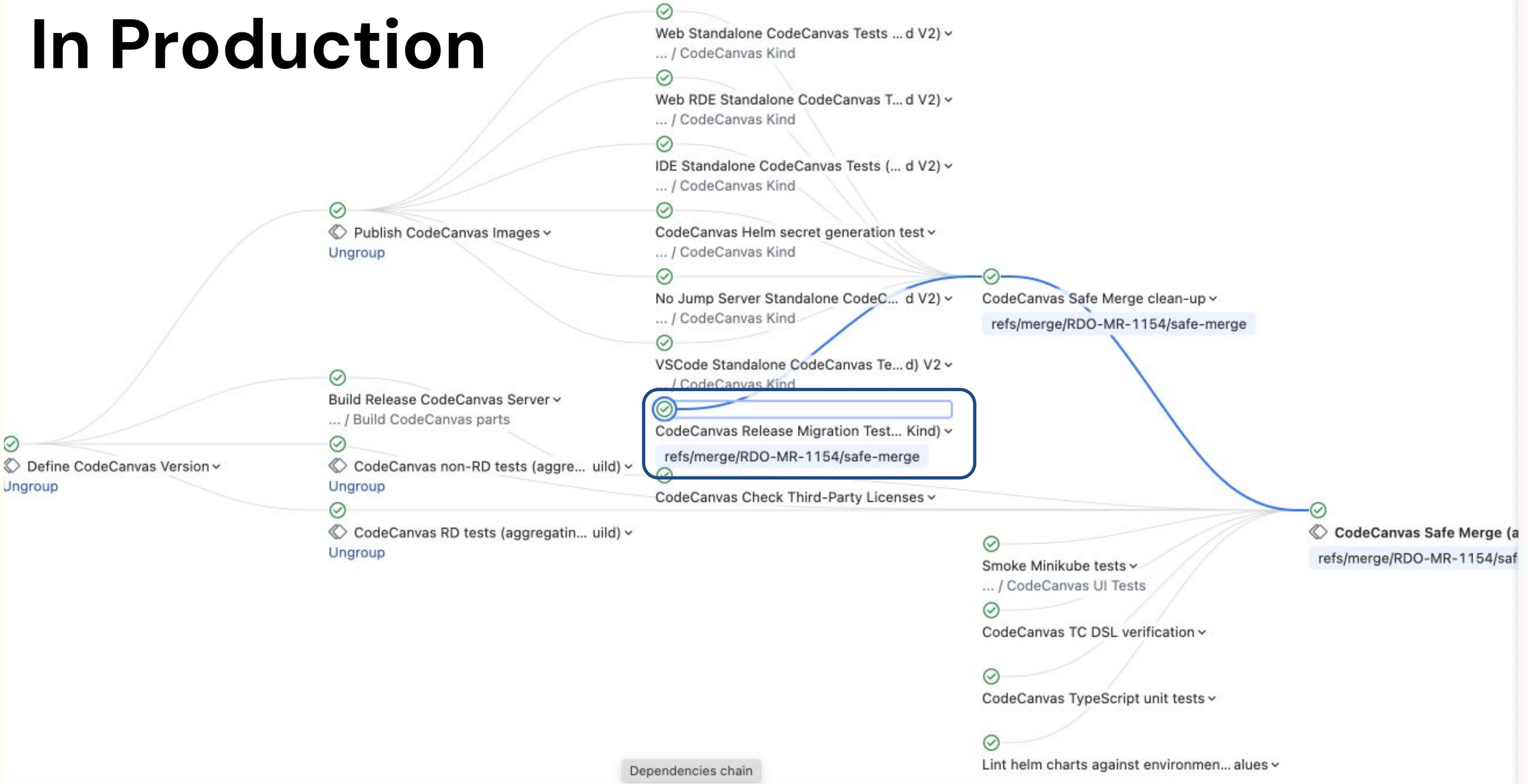
---

avg. 20+ runs / day

2 production bugs caught



In Production



Conclusion

RQ1: Design a continuous **testing framework**

RQ2: Construct a **test suite**

- **RQ2.1:** Which **components** must be tested?
- **RQ2.2:** What are the **expected behaviors and outcomes**?

RQ3: Evaluate **effectiveness**?

System Analysis
Simulate product update
Orchestration-aware

3-step methodology

Dual-coverage strategy

Integrated with CI/CD pipeline

Key Takeaways

For testing Kubernetes-integrated systems, the testing framework must be able to capture the **orchestration logic** of the infrastructure.

For a complex system, constructing a test suite can follow:

1. Identify core components & validation goals;
2. Define test oracles based on structural models.

For Kubernetes-integrated systems under frequent deployments and updates, integrating testing process to CI/CD platform is important.

Bug Example

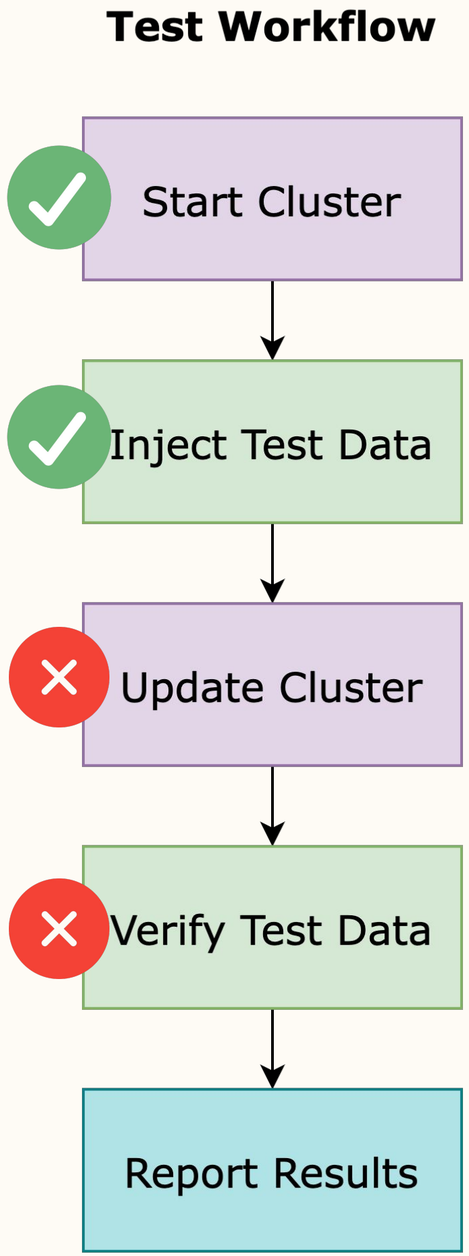
Modification in Helm Chart

```
{{- if .Release.IsInstall }}
```
then create a secret

Helm Upgrade

condition became false
the secret was skipped

Only showed up when testing a **real upgrade behavior**, not from a fresh install



References

[1] CNCF Annual Survey 2024. https://www.cncf.io/reports/cncf-annual-survey-2024/

[2] Astyrakakis, Nikolaos, Yannis Nikoloudakis, Ioannis Kefaloukos, Charalabos Skianis, Evangelos Pallis, and Evangelos K. Markakis. "Cloud-Native Application Validation & Stress Testing through a Framework for Auto-Cluster Deployment." In *2019 IEEE 24th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pp. 1-5. IEEE, 2019.

[3] Turin, Gianluca, Andrea Borgarelli, Simone Donetti, Einar Broch Johnsen, Silvia Lizeth Tapia Tarifa, and Ferruccio Damiani. "A formal model of the kubernetes container framework." In *International Symposium on Leveraging Applications of Formal Methods*, pp. 558-577. Cham: Springer International Publishing, 2020.

[4] Zheng, Tao, Rui Tang, Xingshu Chen, and Changxiang Shen. "KubeFuzzer: Automating RESTful API Vulnerability Detection in Kubernetes." *Computers, Materials & Continua* 81, no. 1 (2024).

[5] Dell'Immagine, Giorgio, Jacopo Soldani, and Antonio Brogi. "Kubehound: Detecting microservices' security smells in kubernetes deployments." *Future Internet* 15, no. 7 (2023): 228.